Legacy Containers and OS Personalities

Carsten Weinhold

Dresden, 2008-12-16
So far:
- Basics Concepts:
  - Tasks, Threads, IPC
  - Memory, Synchronization, ...
- Device Drivers
- Real-Time
- Naming
- Virtualization

Today:
- Legacy Containers / OS Personalities
Motivation

What are legacy containers / OS personalities?
• Emulation environment
• Allow to run legacy applications

How is it different from virtualization?
• Adaptation can be above hardware / OS layer
• Often requires recompilation of the application

Why?
• Reuse existing *applications / APIs*
• Flexibility / Configurability
• Lower resource consumption
• Sometimes easier to achieve
Adaptation to underlying system can happen at various levels:

- **Hardware level (virtualization):**
  - Legacy OS and applications are moved to new OS (full + para virtualization, e.g., L⁴Linux)

- **OS level:**
  - Legacy OS's interfaces are reimplemented on top of new OS (e.g., Wine on Linux)

- **Application level:**
  - Toolkits, frameworks, libraries (e.g., Qt)
Single-server approach:

- All (legacy) system services provided by a single server (e.g., L4Linux)
- No isolation among services
- Provides high level of compatibility, as original implementations are reused
- Ideally, applications do not need to be modified
• Multiple instances of single-server OSes
• Useful for isolation of applications
• Not necessarily identical OS personalities (e.g., have UNIX and Windows servers)
• “Virtual machines”
Hybrid applications (real time):

- “Enlightened” applications:
  - Major parts of applications (e.g., GUIs) do not require real-time guarantees
  - Small parts (e.g., device controller) may need real-time guarantees, can run next to legacy container

- Examples:
  - RT network, RT file system, RT window system / widget
Hybrid applications (security):

- "Enlightened" applications:
  - Most parts are not security critical
  - Small parts require strong isolation from potentially insecure legacy container

- Examples:
  - Cryptographic algorithms and key handling

More in following lectures!
Move from single server to multi server

Challenge: Application wants a consistent view in a distributed system
Central server provides consistent view for both:

- Applications
  - System resources (files, ...)
- Servers
  - Client state (file tables, ...)

Problem:
- Performance bottleneck
- Scalability
- Complexity
Approach: Emulation Library

- Emulation library
  - Per application
  - Interacts with servers
  - Provides consistent view

- Each servers keeps its own client state

- In real world:
  - Applications use C library, emulation library hidden below
  - Emulation provides **POSIX API**
- POSIX “Portable Operating System Interface” is a family of standards (POSIX 1003.*)
- POSIX makes various UNIX-like systems compatible (even Windows NT and newer)
- Defines interfaces and properties:
  - I/O: files, sockets, terminal, ...
  - Threads / synchronization: PThread
  - System tools
  - ...
- POSIX API accessible via C library
• Collection of various common functions
• Abstraction layer above system calls
• Functions with different abstraction levels ...
  – low level: `memcpy()`, `strlen()`
  – medium: `fopen()`, `fread()`
  – high level: `getpwent()`
• ... and dependencies
  – none (freestanding): `memcpy()`, `strlen()`
  – small: `malloc()`: uses `mmap()` or `sbrk()`
  – strong: `getpwent()`: file access, “/etc/passwd”, name service, ...
Multi-Server Aware C Library

- Observations:
  - C library depends on underlying OS
  - Specific implementation may differ

![Diagram of Multi-Server Aware C Library](image)
Example: uClibc on L4

- **What is uClibc?**
  - Reimplementation of C library
  - Designed for Embedded Linux
  - Compatible to GNU C library “glibc”

- **Why uClibc?**
  - Small and portable
  - Optional C++ support available (uClibc++)
  - Licensed under LGPL

- **uClibc + backends provide applications with a consistent view of L4-based multi-server OS**
C Library Backends in L4Env

- C library functions grouped in backends
- Simple example 1: `time`

```c
unsigned int offset;
void l4libc_init_time(void)
{
    int ret;

    l4_calibrate_tsc();
    ret = l4rtc_get_offset_to_realtime(&offset);
    if (ret != 0)
        { /* RTC server not found, */
          /* assuming 1/1/1970, 0:00 */
            offset = 0;
        }
    extern unsigned int offset;

time_t time(time_t *t)
{
    time_t t_temp;
    l4_uint32_t s, ns;

    l4_tsc_to_s_and_ns(l4_rdtsc(), &s, &ns);
    /* get (cached) system time offset */
    t_temp = s + offset;
    if (t)
        *t = t_temp;
    return t_temp;
}
```

1. Initialization:
   Call time server

2. Usage:
   Calculate current time based on time stamp counter (no call to time server necessary)
Multiple implementations for some backends:

- Memory backends:
  - `self_mem`
  - `sigma0_mem, sigma0_pool_mem`
  - `simple_mem`
  - `buddy_slab_mem`
  - `TLSF (real-time memory allocator)`

- I/O backends:
  - `minimal_log_io, io`
Example 2: Minimalist I/O Backend

- Minimalist I/O backend:
  - Simple backend for writing to STDOUT/STDERR
  - Uses L4/Fiasco kernel debugger

```c
#include <unistd.h>
#include <errno.h>
#include <l4/sys/kdebug.h>

int write(int fd, const void *buf, size_t count) __THROW
{
    /* just accept write to stdout and stderr */
    if ((fd == STDOUT_FILENO) || (fd == STDERR_FILENO))
    {
        l4kdb_outnstring((const char*)buf, count);
        return count;
    }

    /* writes to other fds shall fail fast */
    errno = EBADF;
    return -1;
}
```
Example 3: Complex I/O Backend

- Complex I/O backend: L4VFS
- Example:
  1) App calls `open("/dev/tty0")`
  2) Resolve name at name server
     a) Backend calls name server
     b) Name server calls server `con_term`, gets back reference to `tty0`
  3) Backend opens `tty0`, server `con_term` updates client state
  4) Backend updates local file table
  5) Libc returns local file handle
Example 4: POSIX Signals on L4

• Signals used to deliver *asynchronous* event notifications
  – Exceptions (SIGFPE, SIGSEGV, SIGCHLD, ...)
  – Issued by applications (SIGUSR1, SIGUSR2, SIGTERM, ...)

• Signals on Linux
  – built-in kernel mechanism
  – delivered upon return from kernel

• Signals on L4
  – implement in user space
  – map signals to L4 IPC?
  – Problem: IPC is synchronous!
POSIX Signals on L4

Application A

Signal Thread (a.03)

libl4sig

setup signal handlers

Signal Threads

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.03</td>
</tr>
</tbody>
</table>

notify(SIGUSR1)

Application B

Signal Thread (b.04)

libl4sig

setup signal handlers

signal(A, SIGUSR1)

register_handler()
• How signals are dispatched:
  1) Signal thread chooses arbitrary application thread \( E \) (except for L4Env threads)
  2) Stores \( E \)'s thread state (EIP, ESP)
  3) Forces \( E \) to run configured signal handler on dedicated stack
  4) After \( E \) returns from handler: restore original EIP and ESP

• Specialties:
  - SIGSTOP: set all threads to \( \text{l4sleep()} \)
  - SIGALRM: dedicated timer provided by the signal server
  - SIGKILL: signal server kills application at task server
Unnecessary backends can be omitted:
- Applications rely on less components
- Potentially better security properties
- Smaller memory footprint
L4Env offers significant part of POSIX APIs:

- C library: all platform independent parts
  - `strcpy()`, ...
- Dynamic memory allocation
  - Various backends
- I/O support (files, network, terminal, time):
  - L4VFS + L4VFS servers
  - RTC server
- Signal handling:
  - `signal`
- No PThread support (but is available in L4Re)
More Powerful Legacy Containers

- POSIX is limited to basic OS abstractions:
  - No graphics / GUI support
  - No audio support

- Examples for more powerful APIs:
  - SDL “Simple Direct Layer” (focuses on Multimedia)
  - Qt toolkit (focuses on rich GUIs and complete OS abstraction)
• **What is Qt?**
  – Multi-platform toolkit library
  – Provides abstraction of underlying OS
  – Available for:
    • UNIX/X11 (Linux, *BSD, ...)
    • Windows
    • Mac OS X
    • ... and L4Env (+Bastei port mostly done)

• **Qt/L4 based on Qt3 for embedded Linux:**
  – Has its own “Qt Windowing System” (QWS)
  – Relies on POSIX compliant OS
Qt Architecture

Platform Independent

Qt API

File Access
Network
Threads / Synch
GUI
...

Platform Specific

POSIX / L4VFS
POSIX / L4VFS
L4Env / thread, semaphore
QWS

Operating System
• QWS is core part of Qt/L4:
  - Displays and manages windows from multiple Qt applications
  - Operates on shared framebuffer
  - Events distributed via shared memory / sockets
Qt/L4: Features

- Complete support of GUI framework
- Threads, Synchronization supported
- Support for file access
  - Depends on available L4VFS file servers
- Support for network access
  - Basic functionality with L4VFS server “FLIPS”
• Applications (or parts of them) can be developed and tested on Linux
• Rapid Application Development:
  – GUIs can be created with Qt Designer
  – Other tools (e.g., UI compiler) can also usable
DEMO
Coming soon...

• Later today:
  • Practical exercise in E042

• Next year:
  • Big topic “security” starting on January 6
  • 5\text{th} exercise on January 13
References

• Carsten Weinhold: 'Portierung von Qt auf DROPS'
  Großer Beleg 2005

• Resources on POSIX standard:
  http://standards.ieee.org/regauth/posix/

• Previous slide sets for MOS lectures